# Large ACNET Messages

*Rich Neswold, Charlie Briegel, Mike Slicznia k,*
*Dennis Nicklaus, Charlie King*

ACNET has, historically, been a network transport based on datagrams. Migrating to UDP payloads brought the routing abilities of TCP/IP to ACNET, but still kept the datagram limitations, including payload size. ACNET tasks trying to send packets larger than the datagram size limit used their own protocols. Worse still, our primary data acquisition protocols, RETDAT and GETS32, make no attempt at supporting large packets so applications using our standard DAQ libraries can't receive large packets. We wish to design a method to remove this restriction with minimal impact to ACNET tasks (ideally, no change would have to be made to current clients.)

This document describes one possible solution.

---

## Architecture

All ACNET services, if they support large packets, will advertise a task handle of **`LNGMSG`**. Clients do not communicate directly with the **`LNGMSG`** handles, only the **`LNGMSG`** tasks communicate with each other.

When an ACNET client sends data with their request, reply, or USM, the ACNET service determines whether it can be sent via traditional means or if it needs to be handled by the **`LNGMSG`** task. The receiving task will simply see the data (large or small) and has no idea whether it was received directly through its handle or pieced together by its local **`LNGMSG`** task.

---

## **`LNGMSG`** Communications

The **`LNGMSG`** tasks communicate with USMs. The tasks should do whatever validation they can on each packet to make sure the communication state remains uncorrupted.

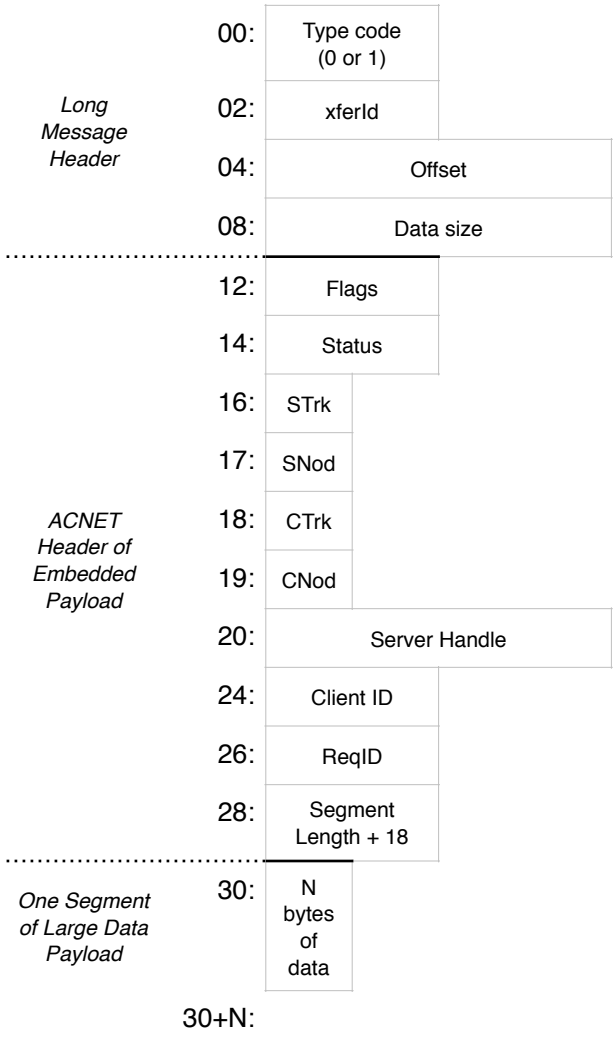
Figure 1 shows the payload of a large reply. This message is sent in the payload of a USM sent from one **`LNGMSG`** task to another. All fields in the **`LNGMSG`** protocol headers are in network byte order.

The large data will be broken into smaller packets[1] and sent in ascending offset order. Each payload will be prepended with the ACNET header of the original

---

[1] We initially chose 8K, but we'll measure performance with other sizes to see if there's a more optimal size.

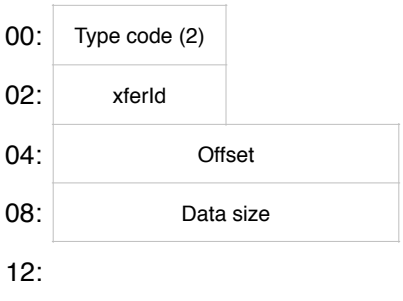
***FIG 1: LONG MESSAGE FORMAT***




request with the slight modification that the message length field will get replaced with the length of the data in the current fragment added to the length of the ACNET header[2]. The type code field will normally be 0, in which case the packet is the next segment of data. If the type code field is 1, it's a segment field, but the sender also wants a resume message from the receiver to determine how to proceed the transfer. The offset and size fields are 32 bits. The size field will always contain the size of the full reply (once it has been pieced back together) and doesn't include any of the ACNET header data. The offset will start at zero and, in each subsequent packet, will equal the previous offset added to the size of the data in the previous packet. The sender will transmit the packets as fast as

[2] For the initial implementation, this field can be used as another validation check for the packet — or it can be ignored. Future enhancements, however, may allow multiple ACNET messages to be sent as a large packet and this length field would help determine where we split the messages apart.

**FIG 2: RESUME MESSAGE**




it can and, after sending a group of segments[3], will ask the receiver where to proceed.

The receiving **LNGMSG** task monitors incoming USMs and does the following:

- If the offset is zero, then a new reply is arriving. The receiver can use the size field to pre-allocate a buffer to hold the rest of the incoming data. After saving the data in the buffer, it sets the next expected offset to be equal to the size of data that was just received.
- If the offset is non-zero, it checks to see if the offset and transfer ID matches a reply that is in progress. If a match is found, the data is appended to the buffer and the next expected offset is updated.
- After appending the data, if the packet also asked for a response (typecode 1 in the long message header), the task will send a resume message (Figure 2) with the current expected offset.
- If the offset is non-zero and a reply to a transfer ID is in progress but the offset is too high (a packet was dropped), the task waits for a packet that also wants a reply. When it arrives, a resume message is sent to the sender with the offset of the missing data.
- When the transfer is complete, the last packet will also require a response. The receiver returns the expected offset (which at this point will be the size of the data) or a previous offset, if a packet was dropped.

Of course, the transfer needs to correctly free resources after the packet is sent or if the request gets cancelled during the transfer.

Due to historic design choices, ACNET constrains packet lengths to even-sizes. We presume that large packets may be used to transfer binary data generated by third party libraries, so they won't follow this convention. The size field should

[3] Initially, we chose 8 packets. Again, performance and reliability measurements may change this number.

show the actual size, odd or not, and the receiver can drop the last byte of the last packet, if it exceeds the size of the data.

A new type code for Level-II diagnostics will be added. The request will return an array of three 32-bit integers indicating: 1) the maximum size of datagrams supported by the node[4], 2) the maximum large packet accepted by the node[5], and 3) the preferred number of packets sent before a query[6].

## Alternate Algorithm

Our primary algorithm for large data transfers is a “sliding window”, where the sender hangs on to the data sent until the receiver confirms its delivery.

These protocol messages, it turns out, can also be used to implement a “bit-map” algorithm. In this algorithm, the sender sends every segment of the large packet and only asks for an acknowledgement after the last packet has been sent. The receiver builds its copy of the large packet with the segments it receives and keeps track of the holes. When the last packet arrives (with the ACK request), the receiver can go back and specify the earliest hole. With each filler sent, the sender always asks for a reply. The receiver replies with each hole’s offset until they are all filled.

This protocol doesn’t allow either participant to enforce an algorithm for an exchange. Mis-matched algorithms will still successfully transfer the data — just at a slightly less efficiency.

[4] Possibly useful. Though it would be interesting to see, first, how UDP size affects the performance. There may be minor speeds gains for large packets at the cost of complexity in querying remote nodes for the “optimal” packet size.

[5] Sending a too-large USM can be dropped silently. A too-large request can result in a “message size error” status. A reply that’s too-large can cause a cancel (with the requesting client getting a message size error — why is the client asking for a message it can’t receive, anyways?)

[6] Mike’s algorithm to adjust the ACK interval based on the network’s behavior is much better than letting the destination node dictate the terms.

## Example Exchanges

A couple of examples may help illustrate how the protocol works. In the following sections, the notation is

```
ACNET_TYPE(ACNET header fields){payload}
```

where `ACNET_TYPE` is the type of ACNET message (USM, REQ, or REP) and is immediately followed by a subset of header fields (not all fields are interesting for these examples.)

In this first example, a 100 Kbyte USM is sent successfully to the server. The sending node asks for a resume message in the third packet. Note the example follows the recommendation to use type code 1 in the first packet.

USM(from:LNGMSG@N1, to: LNGMSG@N2)
{tc:1, xferId:1, offset:0, total:100K, USM(from:CLI@N1, to:SER@N2){25KB of data}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:1, offset: 25K, total:100K}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:0, xferId:1, offset:25K, total:100K, USM(from:CLI@N1, to:SER@N2){*25KB of data*}}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:1, xferId:1, offset:50K, total:100K, USM(from:CLI@N1, to:SER@N2){*25KB of data*}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:1, offset: 75K, total:100K}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:1, xferId:1, offset:75K, total:100K, USM(from:CLI@N1, to:SER@N2){*25KB of data*}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:1, offset: 100K, total:100K}

This next example is a request containing a 100K payload and where the second packet was dropped. The “resume” message recovers the data by requesting the sender restart at the missing offset.

USM(from:LNGMSG@N1, to: LNGMSG@N2)
{tc:1, xferId:2, offset:0, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1) {*25KB of data*}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:1, offset: 25K, total:100K}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:0, xferId:2, offset:25K, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1)
{*25KB of data*}}

*(receiver doesn't get the packet)*

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:1, xferId:2, offset:50K, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1)
{*25KB of data*}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:2, offset: 25K, total:100K}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:0, xferId:2, offset:25K, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1)
{*25KB of data*}}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:0, xferId:2, offset:50K, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1)
{*25KB of data*}}

USM(from:LNGMSG@N1, to:LNGMSG@N2)
{tc:1, xferId:2, offset:75K, total:100K, REQ(from:CLI@N1, to:SER@N2, msgId:1)
{*25KB of data*}}

USM(from:LNGMSG@N2, to:LNGMSG@N1)
{tc:2, xferId:2, offset: 100K, total:100K}

## Recommendations

- The first segment **should** use typecode 1, asking the receiver for a resume message. By doing this, part of the payload gets sent in addition to checking whether the receiver supports large messages (a timeout indicates no support.)
- The last packet of the message **must** use typecode 1 to make sure the entire message was received.
- The sender **may** vary the interval between ACK requests to adapt to network conditions. For instance, the sender might begin the transfer with an interval of 4 packets before asking for an ACK. If there isn't an error, then 8 packets can be sent before the next ACK. If an error occurred, the sender reduces the interval of ACKs.

## Error Conditions

This section tries to address some of the error conditions that can arise and how they should be handled.

*A reply packet gets dropped.*

The next packet will have an offset higher than expected. The receiver stops collecting packets and waits for the next request for a resume message. When asked, the receiver will reply with the next expected offset. The transmission will continue from that offset. If the first packet was dropped, then the offset will be 0.

*The last packet is dropped.*

The sender will set a short time-out waiting for the end-response. If it doesn't receive it, it resends the last packet.

*The sender doesn't resume at the requested offset.*

After sending a resume message, the transfer should eventually restart at the requested offset. If it doesn't in a reasonable time (what is reasonable?), it resends the resume message.

*The ACNET requestor doesn't support large messages.*

The first packet should use typecode 1. If the receiver doesn't reply, it doesn't support large packets.

---

## Unresolved Details

*How does this work with multicasts?*

At first glance, it seems that multicasted requests and USMs should be prevented from participating in large messages. But thinking more about it, multicasting large data could be a killer feature of ACNET. Multicast USMs would probably be better handled by dropping the USM if any packet is dropped. For multicast requests, any of the receivers could ask for a retransmission. All others would see offsets less than their expected offset and would have to patiently wait for the stream to catch back up. Or the sender can send the repeats to the few nodes that needed it. This needs much more discussion.

## Glossary

This section tries to defines some terms used when discussing ACNET communications.

*ACNET Client* — An application that uses the ACNET protocol for network communications. The client is, usually, linked with a library that provides the appropriate interface to the local ACNET service.

*ACNET library* — A library or module used by an application that hides the (possibly) gory details of interfacing with an ACNET service.

*ACNET Service* — The process or task on a machine that implements the actual ACNET protocol (sending and receiving packets, allocating request IDs, routing traffic to clients.) On the CLX machines, the ACNET service is `acnetd`.

*Transfer ID* — A number generated by the **`LNGMSG`** task sending a large packet to tag the transfer. This number should uniquely identify an active transfer.